Apuntes U 2

Tema ***Estructuras de datos***

*"**Los hombres sabios aprenden con los errores que otros cometen; los tontos, con los propios**".*
Henry George Bohn

Definiciones

Estructura: La estructura (del latín structūra) es la disposición y orden de las partes dentro de un todo. También puede entenderse como un sistema de conceptos coherentes enlazados, cuyo objetivo es precisar la esencia del objeto de estudio.

Tipos de estructuras:

- ESTÁTICAS: donde su tamaño es fijo (como ser arrays, cadenas).
- DINÁMICAS: donde su tamaño es variable (como ser pilas, colas, listas).

- Lineal: aquellas en que el almacenamiento se hace en zonas contiguas.
- No Lineal: aquellas en los cuales el almacenamiento puede hacerse en cualquier zona, donde el concepto de "siguiente" es lógico y no físico

Abstracción: Una forma de resolver un problema es analizar las características y condiciones del mismo y de este análisis armar un modelo, el cual nos ayuda para llegar a una solución.

Una forma de resolver un problema es analizar las características y condiciones del mismo y de este análisis armar un modelo, el cual nos ayuda para llegar a una solución.

VENTAJA
Independizar el diseño del programa de la implementación especifica de los datos

Ese modelo es un concepto abstracto, el cual fue creado por nosotros para poder resumir el problema planteado, de una manera que nos resulta manejable.
Los pasos son por medio de acercamientos sucesivos al entorno real.
Partimos desde una visión de alto nivel y luego la vamos refinando añadiendo detalles y consideraciones particulares, hasta que obtenemos una descripción detallada o de "bajo nivel".
Una forma de resolver un problema es analizar las características y condiciones del mismo y de este análisis armar un modelo, el cual nos ayuda para llegar a una solución.

Ese modelo es un concepto abstracto, el cual fue creado por nosotros para poder resumir el problema planteado, de una manera que nos resulta manejable.
Los pasos son por medio de acercamientos sucesivos al entorno real.
Partimos desde una visión de alto nivel y luego la vamos refinando añadiendo detalles y consideraciones particulares, hasta que obtenemos una descripción detallada o de "bajo nivel".

Estructuras Estáticas

* Son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa.
Estas estructuras están implementadas en casi todos los lenguajes.
Ejemplos:
1.- Simples
   a) Boolean
   b) Char
   c) Integer
   d) Real

2.- Compuestas
   a) Arreglos
   b) Conjuntos
   c) Strings
   d) Registros

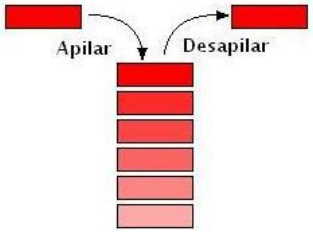
Existen estructuras como las ***pilas***, ***colas*** y ***listas*** que tienen una **implementación** estática y lineal.

Pilas o Stacks

- *Que es una Pila*

Una PILA es una lista lineal de elementos en la cual cada elemento sólo puede ser insertado o eliminado por un extremo denominado CIMA, es decir, los elementos se van a sacar de la pila en orden inverso al que se insertan (pila LIFO, Last In, First Out -ultimo en entrar primero en salir).

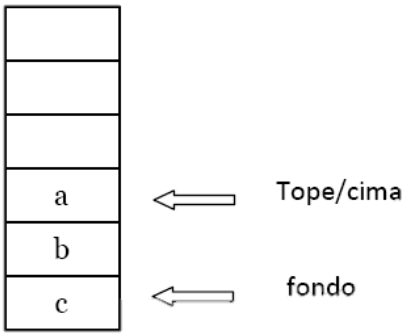
- *Operaciones básicas:*



• insertar: **Apilar**. Insertar un elemento en la cima.
• eliminar: **Desapilar**. Eliminar el elemento de la cima, en orden inverso a la inserción
• buscar: **Cima**. Devolver el elemento que se encuentra en la cima.

- *Otras operaciones:*
- • Determinar si la pila esta vacía
- • Determinar si la pila está llena

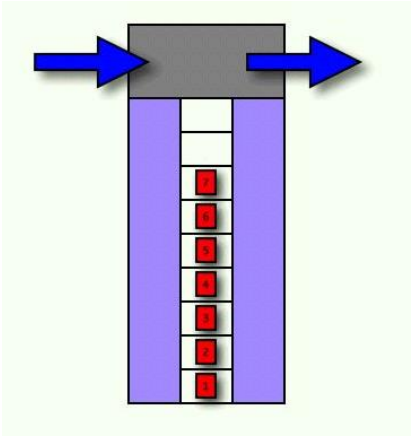
- *Representación*



Las características propias de la Pila nos permite utilizarlas en varias aplicaciones facilitándonos:
• Controlar, desde el Sistema Operativo, la ejecución de todas las ordenes de un archivo batch.
• Recordar al programa principal el punto de llamada de un subprograma y retornar al mismo cuando este termine.
• Formar cadenas de caracteres.
• Separar texto.
• Evaluar expresiones matemáticas.
• Deshacer la última operación realizada, por ejemplo, en un procesador de texto u otros programas similares.
Ahora bien, sobre la PILA podemos realizar las operaciones de:
- PUSH: poner o meter
- POP: sacar
Las pilas pueden ser representadas por medio de una lista unidireccional o por un array lineal.
Representaremos las pilas por medio de un array lineal.
Tendremos un puntero especial apuntando a la CIMA y una variable MAXPILA que determina el tamaño máximo de la pila.
Para controlar el OVER-UNDERFLOW hay que saber determinar el tamaño de la pila, porque si lo hacemos demasiado grande no habrá OVERFLOW pero tendremos una pérdida importante de memoria y viceversa.
Cada vez que un subprograma llama a otro, en la pila del sistema tiene que almacenarse la dirección del programa que llama, a la que se debe regresar cuando el subprograma llamado termina su ejecución.
La dirección de retorno es siempre la siguiente instrucción a la de la llamada.
El 1º en retornar será el último en efectuar la llamada y por eso utilizaremos una pila, dado la particularidad LIFO que mencionamos.

Las pilas y los lenguajes de programación

Las características propias de la Pila nos permite utilizarlas en varias aplicaciones facilitándonos:
• Controlar, desde el Sistema Operativo, la ejecución de todas las ordenes de un archivo batch.
• Recordar al programa principal el punto de llamada de un subprograma y retornar al mismo cuando este termine.
• Formar cadenas de caracteres.
• Separar texto.
• Evaluar expresiones matemáticas.
• Deshacer la última operación realizada, por ejemplo, en un procesador de texto u otros programas similares.



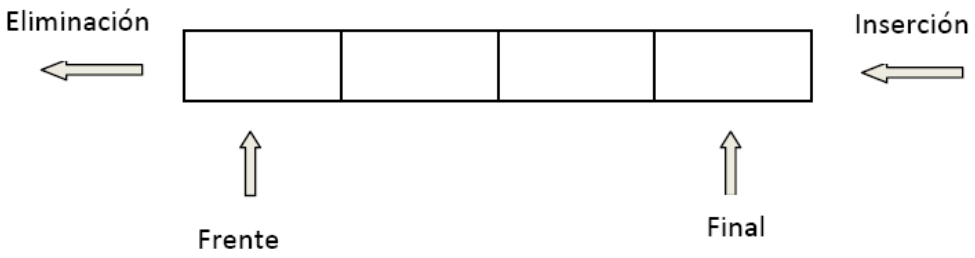
Observaciones:
Las pilas pueden ser representadas por medio de una lista unidireccional o por un array lineal.
Representaremos las pilas por medio de un array lineal.
Tendremos un puntero especial apuntando a la CIMA y una variable MAXPILA que determina el tamaño máximo de la pila.
Para controlar el OVER-UNDERFLOW hay que saber determinar el tamaño de la pila, porque si lo hacemos demasiado grande no habrá OVERFLOW pero tendremos una pérdida importante de memoria y viceversa.

Colas o Queues

Que es una cola
Es otro tipo de estructura de datos, cuyo acceso está restringido. La particularidad de una estructura de datos de cola es el hecho de que sólo podemos acceder al primer y al último elemento de la estructura. Así mismo, los elementos sólo se pueden eliminar por el principio y sólo se pueden añadir por el final de la cola.

Operaciones Básicas:
• Encolar (añadir, entrar, insertar): se añade un elemento a la cola. Se añade al final de esta.
• Desencolar (sacar, salir, eliminar): se elimina el elemento frontal de la cola, es decir, el • primer elemento que entró.
• Frente o primero (consultar, front): se devuelve el elemento frontal de la cola, es decir, el primer elemento que entró.

Representación



Otras Colas

Bicolas:
Son colas en donde los nodos se pueden añadir y quitar por ambos extremos; se les llama DEQUE (Double Ended QUEue). Para representar las bicolas lo podemos hacer con un array circular con Inicio y Fin que apunten a cada uno de los extremos.

Tipos de Bicolas:
• Bicolas de entrada restringida: Son aquellas donde la inserción sólo se hace por el final, aunque podemos eliminar al inicio o al final.
• Bicolas de salida restringida: Son aquellas donde sólo se elimina por el inicio, aunque se puede insertar al inicio y al final.

Otros Tipos
• Colas circulares: en las que el último elemento y el primero están unidos.
• Cola de prioridad: En ellas, los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen.

Colas de Prioridad

Ejemplo: Cuando tenemos un sistema monoprocesador con problemas de paralelismo y concurrencia, podemos ver que la solución es que los procesos que quieran ejecutarse se vayan almacenando en una cola. Esto nos permite que, a medida que acaba un proceso en CPU, comience a ejecutarse el siguiente de la COLA.
Para solucionar esto, a cada proceso se le asigna una prioridad tal que se ejecutarán primero los de prioridad más alta. Inclusive se puede manejar la cantidad de procesos permitidos por prioridad e

inclusive asociar otras variantes a la prioridad, ejemplo tiempo de espero máximo permitido, tiempo de ejecución permitido, uso de memoria, etc.

Entonces, ¿que es una cola de prioridades?
Una cola de prioridades es un conjunto de elementos tal que cada uno se le ha asignado una prioridad de forma que el orden en el que los elementos son eliminados de la cola siguen el siguiente orden:
• Los elementos son procesados por orden de mayor prioridad
• Los elementos que tienen igual prioridad, según el orden en el que llegaron, es decir en forma convencional.

Representación:
- Por medio de una lista unidireccional
- Por múltiples colas

Estructura:
Si la representamos con una lista enlazada, tendrá la siguiente estructura:
- Un campo INFO
- Un campo enlace SIG; que apunta al siguiente elemento de la cola.
- Un campo NPR; que contiene la prioridad de ese elemento.

Tipos
• Con ordenamiento descendente
• Con ordenamiento ascendente

Colas de prioridad ascendente
En las colas de prioridad ascendente (CPA) se pueden insertar elementos en forma arbitraria y solamente se puede remover el elemento de menor prioridad.
Las operaciones que podemos hacer:
- Insert(CPA,x), inserta el elemento x en la cola CPA.
- x=minRemove(CPA) asigna a x el valor del elemento menor (de su prioridad) y se remueve de la cola.

Colas de prioridad descendente
En las colas de prioridad descendente (CPD) es similar, pero sólo permite la supresión del elemento más grande; las operaciones que podemos realizar con este tipo de cola son:
• insert(CPD,x)
• x=maxRemove(CPD)

Con los 2 tipos de colas
La operación empty , la operación insert y la operación borrar, que sólo se aplica si no esta vacía.

Observaciones:
Los elementos de la cola de prioridad no necesitan ser números o caracteres para que puedan compararse directamente, de hecho pueden ser estructuras complejas ordenadas en uno o varios campos.
En las colas de prioridad se pueden sacar los elementos que no están en el primer sitio del extremo donde salen los elementos, dado que el elemento a retirar puede estar en cualquier parte del arreglo.

Cuando se requiere eliminar un dato de una cola de prioridad se necesita verificar cada uno de los elementos almacenados para saber cuál es el menor (o el mayor). Esto trae algunos problemas, el principal es que el tiempo necesario para eliminar un elemento puede crecer tanto como elementos tenga la cola.

Para resolver este problema hay varias soluciones:
Se coloca una marca de "vacío" en la casilla de un elemento suprimido. Este enfoque realmente no es muy bueno, porque de cualquier modo se accesan los elementos para saber si es una localidad vacía o no lo es. Por otro lado, cuando se remueven elementos, se van creando lugares vacíos y después es necesario hacer una compactación, reubicando los elementos en el frente de la cola.

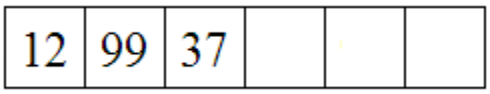
Cada supresión puede compactar el arreglo, cambiando los elementos después del elemento eliminado en una posición y después decrementando en 1. La inserción no cambia. En promedio, se cambian la mitad de los elementos de una cola de prioridad para cada supresión, por lo que esta operación no es eficiente.

Listas (estáticas)

Que es una Lista
Es un conjunto de elementos de un tipo dado que se encuentran ordenados y pueden variar en número. Los elementos en una lista lineal se almacenan en la memoria principal de una computadora en posiciones sucesivas de memoria

Operaciones (en forma genérica):
- Agregar; borrar.
- Recorrer la lista para localizar algún elemento
- Clasificar los elementos en orden ascendente o descendente
- Al conocer la cantidad de nodos y sus ubicaciones, casi no hay restricciones en lo que se puede hacer

Representación



Notas:
Si hablamos de listas en general, no podemos especificar sus características y nos arriesgamos a caracterizarla como dinámica.
Pero si decimos que es una Lista estática, podemos categorizarla como estática y lineal, siendo su implementación mas común en un arreglo unidimensional.

Estructuras Dinámicas

* No tienen las limitaciones o restricciones en el tamaño de memoria ocupada que son propias de las estructuras estáticas.
Mediante el uso de un tipo de datos especifico, denominado puntero (por razones técnicas podremos llamarlo referencia o dato referencial), es posible construir estructuras de datos dinámicas que no poseen nativamente los lenguajes.
Se caracterizan también por el hecho de no saber en que lugar de la memoria se alojaran.

NOTA: Mas allá de sus implementaciones estáticas, las Pilas, Colas y Listas casi siempre se implementan como dinámicas haciendo uso de estructuras con datos y datos referenciales.
Lenguajes como C++ y Pascal tiene un tipo de datos espacial llamado puntero. En Java al no tener ese tipo de datos utilizamos las referencias a los objetos creados (datos referenciales).

¿Que es una referencia?

Las referencias en Java no son punteros ni referencias como en C++. Las referencias en Java son identificadores de instancias de las clases Java. Una referencia dirige la atención a un objeto de un tipo específico. No tenemos por qué saber cómo lo hace ni necesitamos saber qué hace ni, por supuesto, su implementación.

Listas enlazadas - Estructura dinámica

¿ Que es una Lista enlazada o lista ligada?
Una lista enlazada consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior. El principal beneficio

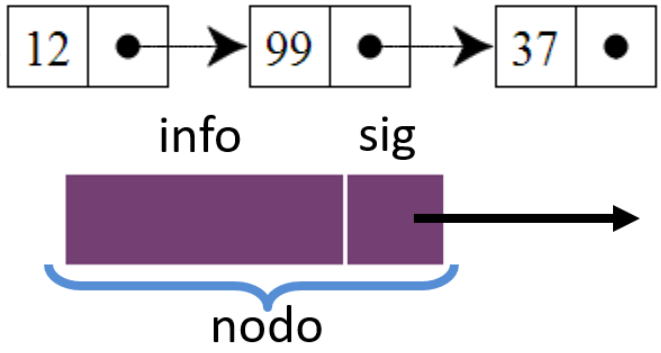
de las listas enlazadas respecto a los vectores convencionales es que el orden de los elementos enlazados puede ser diferente al orden de almacenamiento en la memoria o el disco, permitiendo que el orden de recorrido de la lista sea diferente al de almacenamiento.

Una lista enlazada es un tipo de dato autorreferenciado porque contienen un puntero o enlace (en inglés link, del mismo significado) a otro dato del mismo tipo. Las listas enlazadas permiten inserciones y eliminación de nodos en cualquier punto de la lista en tiempo constante (suponiendo que dicho punto está previamente identificado o localizado), pero no permiten un acceso aleatorio. Existen diferentes tipos de listas enlazadas: listas enlazadas simples, listas doblemente enlazadas, listas enlazadas circulares y listas enlazadas doblemente circulares.

Las listas enlazadas pueden ser implementadas en muchos lenguajes. Lenguajes tales como Lisp y Scheme tiene estructuras de datos ya construidas, junto con operaciones para acceder a las listas enlazadas. Lenguajes imperativos u orientados a objetos tales como C o C++ y Java, respectivamente, disponen de referencias para crear listas enlazadas.

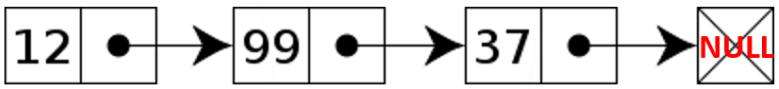
Operaciones:
- Crear: se crea la cola vacía.
- Agregar (añadir, entrar, insertar): se añade un elemento a la lista.
- Borrar (sacar, salir, eliminar): se elimina el elemento de la lista.
- Buscar (consultar, etc): Busca un nodo, comenzando desde el primero y siguendo las referencias.

Representación

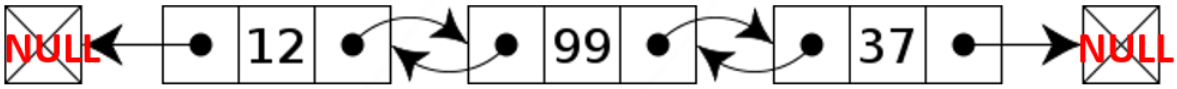


Tipos de Listas enlazadas

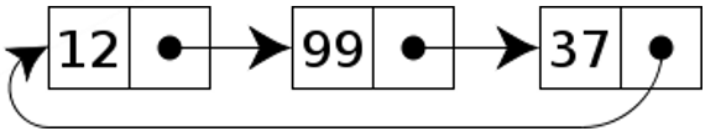
Listas simples enlazadas
Tiene un enlace por nodo. Este enlace apunta al siguiente nodo en la lista, o al valor Nulo o lista Vacía, si es el último nodo.



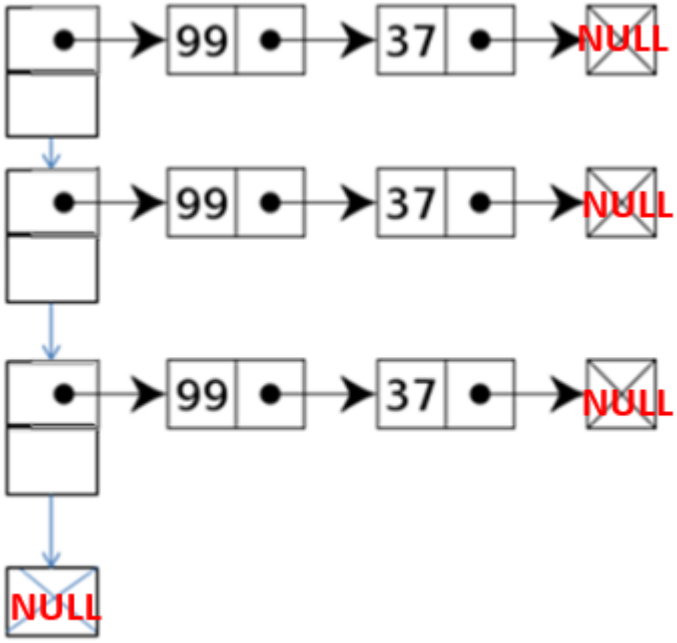
Listas doblemente enlazadas:
Cada nodo tiene dos enlaces: uno apunta al nodo anterior, o apunta al valor Nulo o la lista vacía si es el primer nodo; y otro que apunta al siguiente nodo, o apunta al valor Nulo o la lista vacía si es el último nodo.



Listas enlazadas circulares
En una lista enlazada circular, el primer y el último nodo están unidos. Para recorrer una lista enlazada circular podemos empezar por cualquier nodo y seguir la lista en cualquier dirección hasta que se regrese al nodo original



Listas de listas
El campo de datos de un nodo puede ser otra lista enlazada.



Operaciones de listas

Recorrido
Consiste en visitar cada uno de los nodos que forman la lista.

Inserción
Consiste en agregar un nuevo nodo a la lista (al inicio, antes o después de cierto nodo o Insertar un nodo al final)

Borrado
Consiste en quitar un nodo de la lista, redefiniendo las ligas que correspondan.(el primer nodo, el último nodo, un nodo con cierta información, etc.)

Búsqueda
Consiste en visitar cada uno de los nodos hasta encontrar el elemento dado.

Casos de inserción
1. -Que el nodo insertar pase a ser el primero de la lista. Entonces tendré que variar la variable comienzo.
2. -Que el nodo insertar pase a ser el último de la lista por lo que SIG tendría que pasarlo a NULL
3. -Insertar un nodo a partir de otro concreto.

Pasos a seguir:

A) Estudiar si existe espacio libre en la lista. Si no hay entonces se produciría un OVERFLOW y no puedo hacer la inserción.
B) Crear el nuevo nodo y actualizar los punteros necesarios en esta lista.()
C) Rellenar el nuevo nodo con la información que queremos insertar en él.
D) Reajustar los campos de enlace que sean necesarios en la lista con la que estamos trabajando.

Inserción de un nodo en una lista ordenada
A. -Ver si hay espacio
B. -Si la lista está vacía o si el elemento a insertar es menor que el primero de la lista aplico el algoritmo 1.
C. -Si no se cumplen las condiciones anteriores tendré que localizar la posición LUG que le corresponde al nodo e insertarlo allí con el algoritmo 3 o 2 (si es el último).

Casos de Borrado
1. - Que el nodo a borrar sea el primero de la lista.
2. - Borrar un nodo conocido su predecesor.
   Caso particular: borrar un nodo que está el final de la lista
3. - Borrado de un nodo que contiene un determinado elemento de información.

Consideraciones

Nodos de referencia o nodos centinelas: A veces las listas enlazadas tienen un nodo centinela (también llamado falso nodo o nodo ficticio) al principio o al final de la lista, el cual no es usado para guardar datos. Su propósito es simplificar o agilizar algunas operaciones, asegurando que cualquier nodo tiene otro anterior o posterior, y que toda la lista (incluso alguna que no contenga datos) siempre tenga un "primer y último" nodo. (algunos lo llaman cabecera y tierra)

Overflow y Underflow (en estructuras dinámicas)

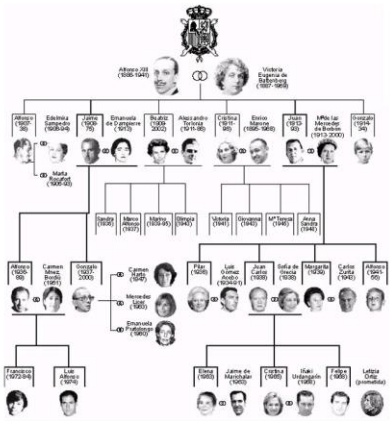
Una lista sufre un OVERFLOW cuando se intenta insertar un elemento en dicha lista estando la lista de espacio disponible vacía, es decir, cuando no hay más memoria libre.

Una lista sufre UNDERFLOW cuando intentamos borrar un elemento de la lista estando ésta vacía.

Estas 2 situaciones habrá que controlarlas siempre que hagamos una inserción y siempre que hagamos un borrado.

Arboles

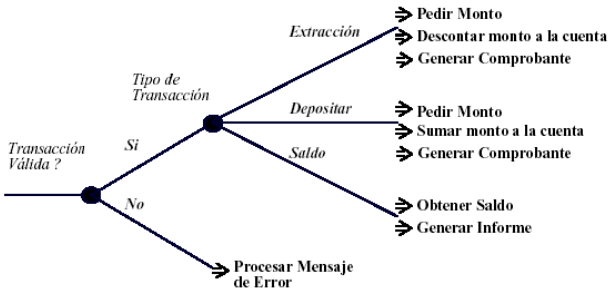
El inconveniente de las estructuras de datos dinámicas lineales es que cada elemento sólo tiene un elemento siguiente, es decir, sólo puedo moverme una posición.
Los árboles representan estructuras no lineales y dinámicas.
Un árbol se define como un conjunto finito de elementos llamados nodos que guardan entre ellos una relación jerárquica.
Existe un nodo diferenciado llamado raíz del árbol, y los demás nodos forman conjuntos diferentes cada uno de los cuales es a su vez un árbol, a estos árboles se los denominó subárboles.
Del nodo raíz van a salir las ramas que van conectando las ramas inferiores.

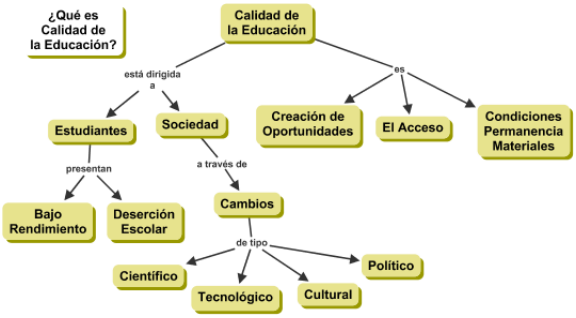


Un árbol vacío es aquel que no tiene ningún nodo.

Con esta estructura de datos podemos realizar varias aplicaciones, como ser:

- Registrar la historia de eventos
- Árboles genealógicos
- Análisis de circuitos eléctricos
- Fórmulas matemáticas
- Numerar capítulos y secciones de un libro





Conceptos

NODO: Es cada uno de los elementos del árbol
RAÍZ: Es un nodo especial del que descienden todos los demás nodos. Este nodo es el único que no tiene "padre".
HOJA (o nodo terminal): Es donde termina el árbol, no tiene ningún descendiente (nodos
HIJO: Cada nodo que no es hoja y tiene debajo de él 1 o varios nodos
PADRE: Todo nodo excepto la raíz, tiene asociado un ÚNICO nodo predecesor al que llamaremos padre.
HERMANO: Son los nodos que son hijos de un mismo padre .
NODO INTERNO: Cualquier nodo que no sea una hoja.

NIVEL: Cada nodo de un árbol tiene asignado un número de nivel superior en una unidad a su padre.
CAMINO: Una sucesión de enlaces que conectan dos nodos.
RAMA: Camino que termina en una hoja.
PROFUNDIDAD (altura) DE UN ÁRBOL: Es el máximo número de nodos de una rama
PESO DE UN ÁRBOL: Número de nodos de un árbol.
BOSQUE: Colección de 2 o más árboles.

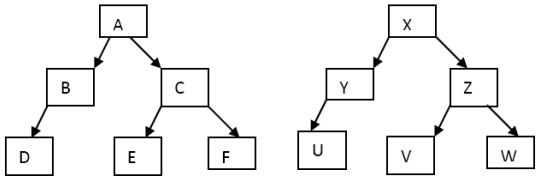
Arboles Binarios

¿Qué es un Árbol binario?

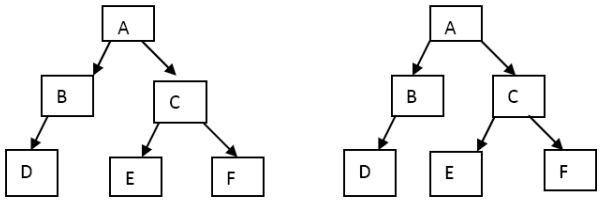
Un árbol binario es una estructura de datos en la cual cada nodo siempre tiene un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a null, es decir que no almacena ningún dato, entonces este es llamado un nodo externo. En el caso contrario el hijo es llamado un nodo interno. Usos comunes de los árboles binarios son los árboles binarios de búsqueda, los montículos binarios y Codificación de Huffman.

Tipos de Árboles

Árboles binarios similares



Árboles binarios equivalentes o copias



Árboles binarios equilibrados:
Sus alturas, profundidades o pesos (dependiendo el criterio) se diferencian como máximo en 1 unidad.

Árboles binarios llenos
Un árbol binario lleno es un árbol en el que cada nodo tiene cero o dos hijos, es decir su factor de equilibrio es 0.

Árboles binarios perfecto:
Un árbol binario perfecto es un árbol binario lleno en el que todas las hojas (vértices con cero hijos) están a la misma profundidad (distancia desde la raíz, también llamada altura).

Representación de árboles binarios en memoria

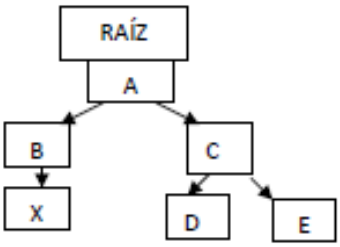
- Representación enlazada o de punteros
- Representación secuencial o con arrays

El principal requerimiento para cualquier representación es que para cualquier árbol T se tenga acceso directo a la raíz de dicho árbol y dado un nodo N del árbol se tenga acceso directo a sus hijos.

Representación enlazada o de punteros

El árbol se representa por medio de una lista enlazada especial en la que cada nodo tiene 3 campos:
- IZQ; campo enlace o puntero al hijo izquierdo del nodo, si dicho hijo no existe, contiene NIL.
- DER; campo enlace o puntero al hijo derecho.
- RAIZ; variable que apunta al nodo raíz del árbol.

Un árbol estará vacío cuando RAIZ <--- NIL.
Para las inserciones y borrados utilizaremos la lista DISP, va a ser una lista enlazada normal en la que cada nodo tendrá los 3 campos siendo el IZQ el campo enlace.

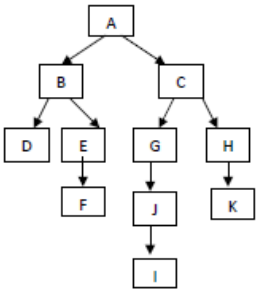


A B C X D E

Representación secuencial

Con 3 arrays: INFO, DER y IZQ y una variable puntero RAIZ.
Cada nodo vendrá representado por una posición K tal que la posición K de INFO, INFO[K], contiene la información de K.
IZQ[K] --> contiene la posición del array donde está almacenado el hijo izquierdo de K.
DER[K] --> contiene la posición del array en la que está almacenado el hijo derecho de K.
Para las inserciones y borrados utilizaremos la lista DISP que enlazará las posiciones del array. La variable puntero DISP apuntará a la primera posiciones libre del array y el resto de las posiciones se enlazarán entre si mediante el campo IZQ.

Con 1 array: sea T un árbol binario lleno a casi lleno, para este tipo de árbol lo más eficiente es utilizar un único array para almacenarlo siguiendo las siguientes normas:
El array lo llamaremos **ARBOL** y en **ARBOL**[1] meteremos la raíz del árbol.
Para cada nodo N que ocupe la posición K del árbol, su hijo izquierdo se encuentra almacenado en la posición (2 * K) y su hijo derecho en la posición (2 * k + 1).
Para poder mantener un árbol con esta representación si la profundidad del árbol es h el array en el que se almacena tiene que tener como mínimo 2 h -1 posiciones porque ése es el númeromáximo de nodos que puede tener un árbol lleno de profundidad h.
El inconveniente de esta representación es que si no está lleno se pierde mucho espacio y que además no podemos insertar bajando de nivel porque no hay profundidad.



Profundidad = 5
2 5 - 1 = 31 posiciones
A B C D E G H F J K I

Recorrido de árboles binarios

Recorrido de un árbol es el proceso que consiste en acceder una sola vez a cada nodo del árbol; hay 3 maneras estándar de recorrer un árbol y en las tres se dan los mismos pasos y también coinciden en que se recorre primero su árbol izquierdo y luego el derecho.
Las tres maneras se diferencian en el momento que se accede al nodo raíz.

1. – PREORDEN: 1º se procesa la raíz.
2º subárbol izquierdo.
3º subárbol derecho.

2. – INORDEN: 1º subárbol izquierdo.
2º se procesa la raíz.
3º subárbol derecho.

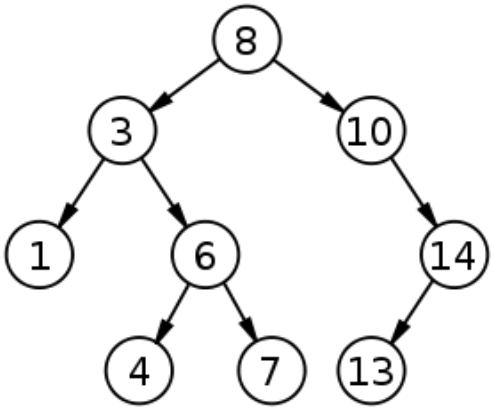
3. – POSTORDEN: 1º subárbol izquierdo.
2º subárbol derecho.
3º se procesa la raíz

Árboles binarios de búsqueda

Un árbol binario de búsqueda también llamados BST (*acrónimo del inglés* ***Binary Search Tree***) es un tipo particular de árbol binario en el que los elementos están organizados de tal manera que facilitan las búsquedas

El árbol depende de un campo **clave** que es distinto para cada nodo y está ordenado respecto a ese campo, tal que si T es un árbol binario, es de búsqueda si cada nodo N del árbol tiene la siguiente propiedad: "*El campo* ***clave de n*** *es mayor que cualquiera de los campos clave de los nodos del subárbol izquierdo de n y de la misma manera el campo clave de n es menor el campo clave de todos los nodos que se encuentran en el subárbol derecho de n"*

Esto supone que si hago un recorrido **inorden** del árbol obtendremos sus elementos ordenados por el campo clave en orden ascendente.



NOTA: Para una fácil comprensión queda resumido en que es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores.

Dos árboles binarios de búsqueda *pueden no ser iguales* aun teniendo la misma información, va a depender del orden en el que ingrese la información.

- *Búsqueda de un elemento*

El algoritmo localiza la posición PTR del nodo que contiene el valor del elemento y en PAD dejamos la localización del nodo padre de PTR, la filosofía del algoritmo consiste en comenzar comparando el elemento con la raíz del árbol y luego ir recorriendo dicho árbol por la derecha o por la izquierda según si el elemento es mayor o menor que el nodo que estamos examinando y así hasta encontrar el elemento o llegar a una hoja del árbol.

- *Inserción de un elemento*

1º Mirar si hay espacio disponible en la lista DISP.

2º Ver si el elemento está en el árbol y si no está, localizar la posición dónde debe ser insertado, para lo que utilizaré el algoritmo de búsqueda tal que PAD será el nodo al que hay que enganchar el nodo que insertamos.

3º Averiguar si el debe insertarse como el hijo derecho o izquierdo de PAD.

4º Finalmente, el nodo tiene que quedar como una hoja.

- *Eliminación de un elemento*

Con el elemento lo primero que hay que hacer es controlar la situación de *UNDERFLOW*, después buscar el nodo y una vez encontrado el nodo se pueden dar 3 casos:

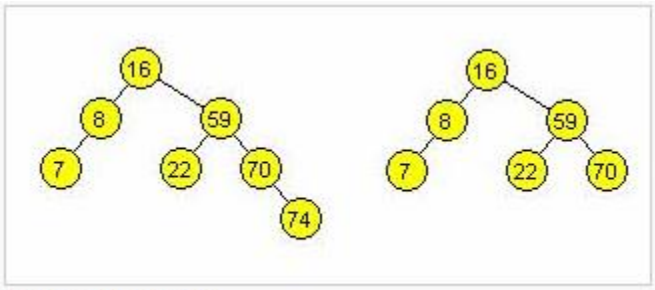
1. - que el nodo no tenga hijos: simplemente se borra y se establece a nulo el apuntador de su padre.

2. - que el nodo tenga un hijo: se borra el nodo y se asigna su subárbol hijo como subárbol de su padre.

3. - que tenga dos hijos: la solución está en reemplazar el valor del nodo por el de su predecesor o por el de su sucesor en inorden y posteriormente borrar este nodo. Su predecesor en inorden será el nodo

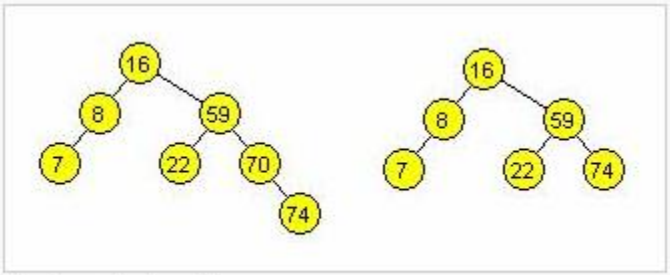
más a la derecha de su subárbol izquierdo (mayor nodo del subarbol izquierdo), y su sucesor el nodo más a la izquierda de su subárbol derecho (menor nodo del subarbol derecho).

- *Borrado sin hijos*



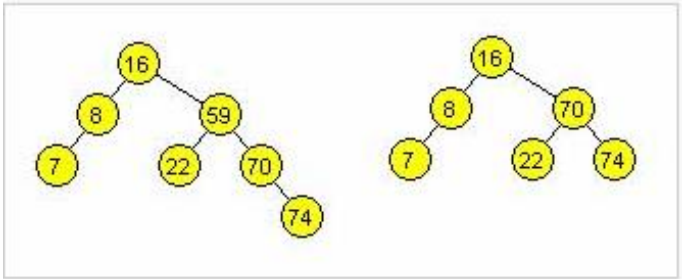
Nodo a eliminar 74

- *Borrado con un hijo*



Nodo a eliminar 70

- *Borrado con 2 hijos*



Nodo a eliminar 59

Aplicaciones de Árboles Binarios

- *Reconocimiento de patrones*

Una de las aplicaciones más importantes de los árboles es utilizarla en el reconocimiento de patrones que es una de las ramas de la inteligencia artificial y en general en la mayoría de las aplicaciones en que se necesita hacer búsquedas.
La filosofía de reconocimiento de patrones es llegar a una conclusión al ir moviéndonos en el árbol según unas determinadas pistas.
Se pueden representar muchos patrones en forma de árboles binarios. El truco está en asignar los significados adecuados a los movimientos de ir hacia la izquierda o la derecha.
El inconveniente es que como lo que usamos son árboles binarios, estas relaciones de búsqueda o patrones sólo pueden ser binarias.

- *Transformación de una notación en otra*

La transformación de una notación a otra se puede realizar estructurando los exponentes como un árbol binario tal que un recorrido sea notación infija, un recorrido preorden la notación prefija y uno postorden la postfija.
La primera tarea es construir el árbol teniendo en cuenta que cada subárbol tendrá como raíz un operador y los hijos o son operados u otros subárboles.
Para equilibrar 0 + a (b + c * d)

Consideraciones de Complejidad

Se definen dos operaciones: Insertar un elemento en el conjunto, y otra que busca y descarta (selecciona) el elemento con valor menor del campo prioridad.

Es necesario encontrar una nueva estructura, ya que las vistas anteriormente tienen limitaciones.
Una lista ordenada en forma ascendente por la prioridad, permite seleccionar el mínimo con costo **O(1)**.
Pero la inserción, manteniendo en orden tiene costo promedio **O(n)**; si encuentra el lugar para insertar a la primera es costo 1; pero si debe recorrer toda la lista, debe efectuar n comparaciones; en promedio debe efectuar un recorrido de n/2 pasos.

Una lista no ordenada, tiene costo de inserción **O(1)**, y búsqueda **O(n)**.

En ambos casos la repetición de n operaciones sobre la estructura da origen a complejidad n^2.

Estudiaremos usar la estructura de un árbol binario, ya que ésta garantiza que las operaciones de inserción y descarte sean de complejidad $O(\log_2 n)$, pero deberemos efectuar modificaciones ya que en una cola de prioridad se permiten claves duplicadas.
Las operaciones en la estructura Colas de Prioridad tienen una complejidad Logarítmica.

Una búsqueda binaria sin éxito se comporta como $O(\mathrm{N} \log_{\square} n)$

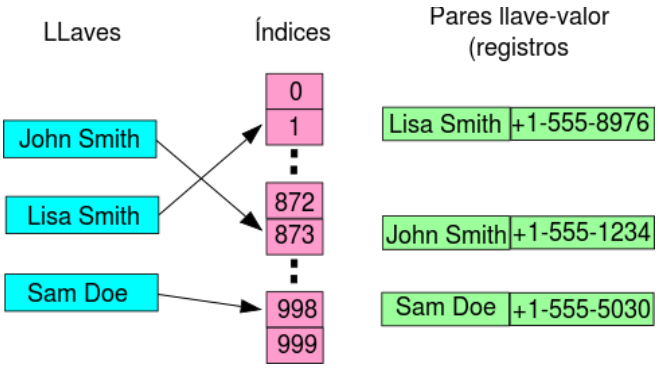
Una búsqueda secuencial sin éxito el algoritmo se comporta como $O(N)$
Crecimiento de los números armónicos responde a una complejidad Logarítmica

Tablas Hash

Una tabla Hash es un contenedor asociativo (*tipo* ***Diccionario***) que permite un almacenamiento y posterior recuperación eficientes de elementos (*denominados* ***valores***) a partir de otros objetos, llamados **claves o llaves**.

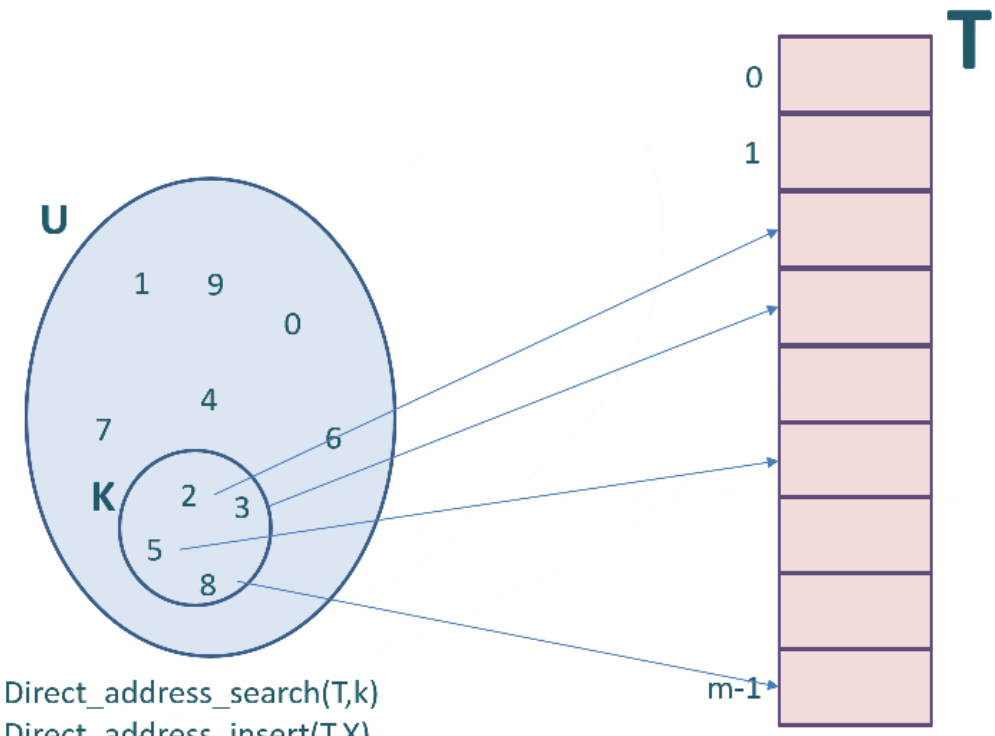
Una tabla hash se puede ver como un conjunto de entradas. Cada una de estas entradas tiene asociada una clave única, y por lo tanto, diferentes entradas de una misma tabla tendrán diferentes claves. Esto implica, que una clave identifica unívocamente a una entrada en una tabla hash. Por otro lado, las entradas de las tablas hash están compuestas por dos componentes:

- La propia clave
- La información que se almacena en dicha entrada.



Ejemplo: Se necesita organizar los periódicos que llegan diariamente de tal forma que se puedan ubicar de forma rápida, entonces se hace de la siguiente forma - se hace una gran caja para guardar todos los periódicos (*una tabla*), y se divide en 31 contenedores (*ahora es una "**hash table**" o tabla fragmentada*), y la clave para guardar los periódicos es el día de publicación (*índice*). Cuando se requiere buscar un periódico se busca por el día que fue publicado y así se sabe en que zócalo (*bucket*) está. Varios periódicos quedarán guardados en el mismo zócalo (*es decir **colisionan** al ser almacenados),* lo que implica buscar en la sub-lista que se guarda en cada zócalo. De esta forma se reduce el tamaño de las búsquedas de **O(n)** a, en el mejor de los casos, **O(1)** y, en el peor, a **O(log(n)).**

Direccionamiento directo (No es una Tabla Hash)

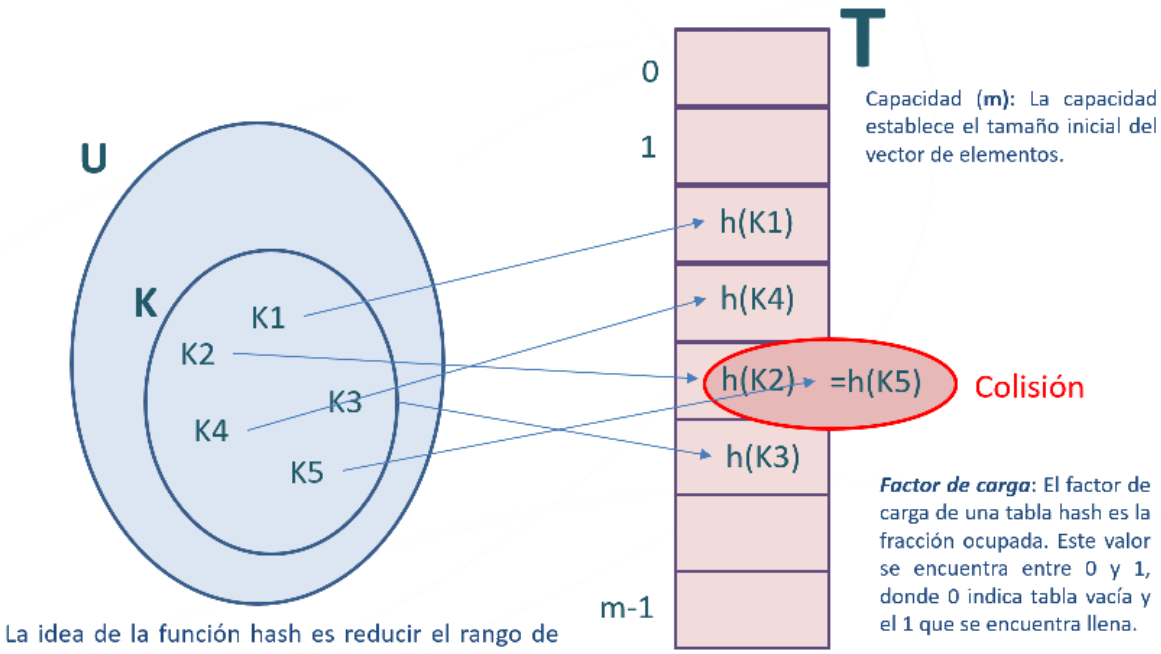


Direct_address_search(T,k)
Direct_address_insert(T,X)
Direct_address_Delete(T,X)

Operaciones de Tiempo Constante **O(1)**

Para representar conjuntos dinámicos, se usa un arreglo o un tabla de direccionamiento directo T[0..m – 1], en la cual cada posición, o slot, corresponde a una llave en el universo de llaves U.

Direccionamiento con Tablas Hash



Objetivo

Definir una función que, dada una clave de búsqueda, determine la posición de almacenamiento del ítem (de datos), similar a la funcionalidad de un "diccionario": teniendo una clave, se accede a un valor (clave, valor) para intentar reducir el tiempo a constante: O(1)

Características

- *No existe un ordenamiento físico de los datos.*
- *Facilita inserción y eliminación rápida de registros por clave.*
- *Encuentra registros por búsqueda asociativa con escasos accesos en promedio.*
- *La teoría de las tablas hash se basa en probabilidades.*

Para implementar una Tabla Hash se necesita

- *Una estructura de acceso (un arreglo).*
- *Una estructura de datos con una clave (datos propiamente dichos).*
- *Una función (función hash).*

Algunas Características

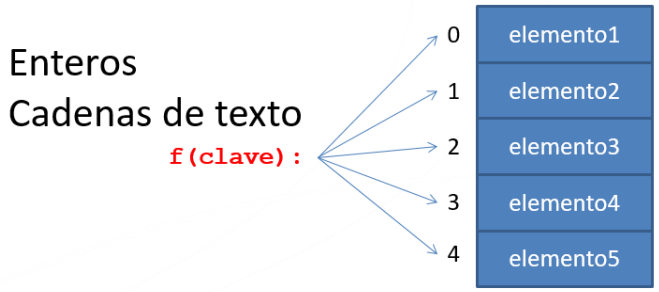
El tiempo medio de recuperación de información es constante, es decir, no depende del tamaño de la tabla ni del número de elementos almacenados en la misma.

Una tabla hash está formada por un **array de entradas**, que será la estructura que almacene la información, y por una **función de dispersión**. Esta función permite asociar el elemento almacenado en una entrada con la **clave** de dicha entrada; por lo tanto, es un algoritmo crítico para el buen funcionamiento de la estructura.

Es frecuente que se produzcan **colisiones**, que se generan cuando para dos elementos de información distintos, la *función de dispersión* les asigna la misma *clave*.

Función Hash

Es una operación que consiste en transformar una clave en una posición dentro de la tabla Hash



Es posible que diferentes claves apunten a la misma posición de la tabla provocando **COLISIONES**

- *Función Hash, Modulo*

hash(llave) = llave mod Max
Donde Max es el tamaño de la tabla

Inconvenientes:
-sólo se usa la parte menos significativa de la llave
-puede producir distribuciones muy poco homogéneas
- cuando alguna de las "terminaciones" de las llaves tiende a
repetirse más que otras

- *Función Hash, troceado en grupos de cifras*

$$\text{hash(llave)} = \left(\sum_{i=0}^{n-1} \left(\frac{llave}{max^i} \bmod Max \right) \right) \bmod Max$$

+Ventaja: utiliza todas las cifras de la llave
-Inconveniente: es algo más costosa de calcular

- *Función Hash, suma de los códigos ASCII de los caracteres*

hash(llave) = $\left(\sum_{i=0}^{length-1}\left(ascii(llave_i)\right)\right) mod\ Max$

Donde Max es el tamaño de la tabla

+Ventajas: sencilla y eficiente
-Inconveniente: las llaves con los mismos caracteres (aunque estén en distinto orden) generan el mismo código de dispersión

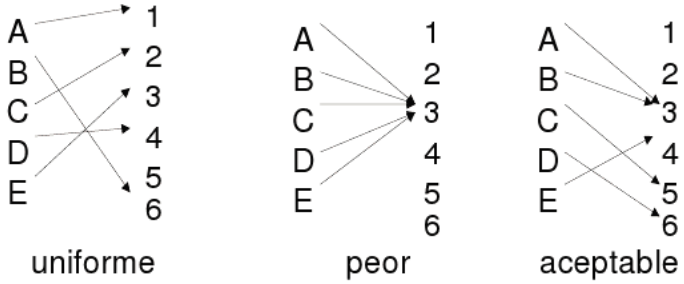
- *Función Hash, suma con pesos de los códigos ASCII de los caracteres*

hash(llave) = $\left(\sum_{i=0}^{length-1}\left(ascii\left(llave_i * peso^i\right)\right)\right) mod\ Max$

+Ventaja: el código depende de los caracteres que forman la llave y también del orden que ocupan dichos caracteres
-Inconveniente: más costosa de calcular

Colisiones

Una colisión de hash es una situación que se produce cuando dos entradas distintas a una función de hash producen la misma salida.
Es matemáticamente imposible que una función de hash carezca de colisiones, ya que el número potencial de posibles entradas es mayor que el número de salidas que puede producir un hash. Sin embargo, las colisiones se producen más frecuentemente en los malos algoritmos. En ciertas aplicaciones especializadas con un relativamente pequeño número de entradas que son conocidas de antemano es posible construir una función de hash perfecta, que se asegura que todas las entradas tengan una salida diferente. Pero en una función en la cual se puede introducir datos de longitud arbitraria y que devuelve un hash de tamaño fijo (como MD5), siempre habrá colisiones, debido a que un hash dado puede pertenecer a un infinito número de entradas.



Dispersión/Colisión: dos o más elementos tienen la misma posición:

Hash cerrado (Direccionamiento Abierto):

- Exploración lineal,

- Exploración cuadrática

Hash abierto (Direccionamiento Cerrado):

- Encadenamiento separado

Características de la Función HASH

- **Cálculo rápido**: Una buena función hash es simple, se puede calcular rápidamente, su principal ventaja es su velocidad. Si la función hash es lenta, esta velocidad será degradada. El propósito de una función hash es tomar una serie de valores y transformarlos en los valores de índice, de tal manera que los valores de clave están distribuidos al azar a través de todos los índices de la tabla hash. Las claves pueden ser totalmente aleatorias o no.

- **Claves aleatorias**: Llamamos a una función de hash "perfecta" si se mapea cada clave en una ubicación de tabla diferente. En la mayoría de los casos no existe esta situación, ya que la función de hash va a necesitar comprimir una gama más amplia de claves en un rango menor de números de índice que coincide con el rango del arreglo. Para lograr esto aplicamos el modulo a la función de hash para que de un valor dentro de ese rango:

  ndice =indice% arraySize

  Se trata de una sola operación matemática, y si las claves son realmente aleatorias, los índices resultantes serán aleatorios, y por lo tanto bien distribuidas.

- **Claves no Aleatorias**: En este caso los datos se distribuyen de manera no aleatoria. Imagínese que se usan como clave los números de autopartes. Estos números van a tener un formato particular, por ejemplo: 033-400-03-94-05-0-535 donde cada uno de los valores tiene una interpretación y un rango de valores posibles para cada uno. Por ejemplo, el 033 es un valor que representa una categoría cuyo valor posible es de 000 a 050. Estas claves no están distribuidas al azar.

- **Usar un número primo para el modulo**: Generalmente, la función de hash implica el uso del operador de módulo (mod o %) con el tamaño de la tabla. Además como veremos en la próxima sección, es importante que el tamaño de la tabla sea un número primo cuando utilizamos resolución de colisiones por *exploración lineal* o *cuadrática*. Sin embargo, si las claves no se distribuyen de manera aleatoria, es importante que el tamaño de la tabla se defina con un *número primo* independientemente del tipo de función de hashing que se utilice. Esto es así dado que si muchas claves comparten como divisor el tamaño del arreglo, puede tender a ubicar los elementos en la misma ubicación, causando la agrupación. Usando un número primo como tamaño de la tabla se elimina esta posibilidad.

Tipos de Hashing

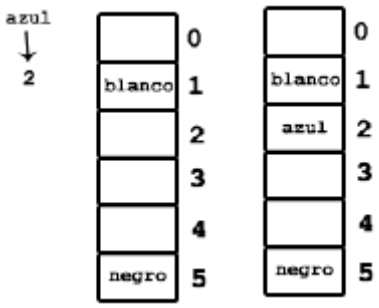
- ***Hashing Perfecto***: Existe una Función de Enumeración que asigna a cada valor del dominio una única posición de memoria. No posee colisiones.
- ***Hashing Puro***: La función de Hash puede asignar a dos valores distintos el mismo valor hash. Estos dos valores reciben el nombre de sinónimos. Las estructuras de hashing puros poseen

colisiones y en consecuencia se deberán establecer mecanismos para tratar los mismos. Podemos clasificarlos en estructuras cerradas y abiertas y dentro de las abiertas en estáticas y dinámicas:

- **Cerradas**: No utilizan un nuevo espacio en memoria.
- **Abiertas**: Utilizan espacio adicional.
  - ***Estática***: La estructura principal no crece.
  - ***Dinámica***: La estructura principal se expande a medida que aumenta la cantidad de elementos.

Tablas HASH (Operaciones Básicas)

Inserción: El proceso de inserción en una tabla hash es muy simple y sencillo. Sobre el elemento que se desea insertar se aplica la función de dispersión. El valor obtenido tras la aplicación de esta función será el índice de la tabla en el que se insertará el nuevo elemento.
Veamos este proceso con un ejemplo. Sobre la siguiente tabla hash se desea introducir un nuevo elemento, la cadena azul. Sobre este valor se aplica la función de dispersión, obteniendo el índice 2.

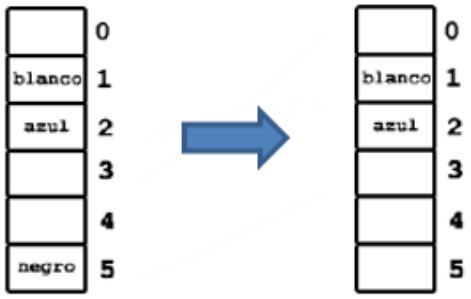


En el caso de que se produzca una ***colisión*** al tratar de insertar el nuevo elemento, el procedimiento será distinto en función del tipo de hash con el que se esté tratando, y se deberá tratar según la forma de resolución de colisiones implementada.

Búsqueda: Se implementa con la función de Hash en forma directa, a no ser que existan colisiones. En tal caso se deberá usar la técnica propuesta sobre los valores colisionados.

Borrado: El borrado en una tabla hash es muy sencillo y se realiza de forma muy eficiente.

Una vez indicada la clave del objeto a borrar, se procederá a eliminar el valor asociado a dicha clave de la tabla.
Esta operación *se realiza en tiempo constante, sin importar el tamaño de la tabla o el número de elementos que almacene* en ese momento la estructura de datos. Esto es así ya que al ser la tabla una estructura a la que se puede acceder directamente a través de las claves, no es necesario recorrer toda la estructura para localizar un elemento determinado.
Si sobre la tabla resultante de la inserción del elemento azul realizamos el borrado del elemento negro, la tabla resultante sería la siguiente:



Otras Operaciones: Una de las principales operaciones que se pueden realizar en las tablas hash es la ***redispersión***. Se suele realizar cuando el factor de carga (número de elementos / capacidad de la tabla) de la tabla supera cierto umbral.

La ***redispersión*** consiste en pasar todos los elementos de la tabla original a una nueva tabla de un tamaño mayor. De esta forma, se reduce el factor de carga de la tabla.

Resolución de colisiones

Hash cerrado (Direccionamiento Abierto):
- **Exploración lineal**
- **Exploración cuadrática**
- **Doble Hasheo**

Hash abierto (Direccionamiento Cerrado):
- **Encadenamiento separado**

Tablas HASH . Protección Activa

Es cuando se intentan evitar las colisiones diseñando buenas funciones hash

- *Función hash básica para claves enteras*

Buena opción: f(c) = c % B, siendo c la clave y B el tamaño de la tabla
Colisiones: n / B, siendo n el número de elementos y B el tamaño de la tabla

- *Función hash básica para claves string*

Hay que convertir la cadena a un valor numérico y luego aplicarle la función hash para claves enteras Ej. utilizando el código ASCII de cada carácter

Asignación de pesos para incrementar el rango
Una ponderación típica es 27^{2-i} siendo i la posición del carácter en la cadena

33	!	34	"	35	#	36	$	37	%	38	&	39	'	
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~		

Letra	Código ASCII	Código ponderado
P	80	$80 * 27^2 = 58320$
u	117	$117 * 27^1 = 3159$
e	101	$101 * 27^0 = 101$
r	114	
t	116	
a	97	
TOTAL:	**625**	**61580**

Asignación de pesos para incrementar el rango

Se puede optimizar la multiplicación utilizando como peso 32

Letra	Código ASCII	Código ponderado	Código ponderado (32)
P	80	$80 * 27^2 = 58320$	$80 * 32^5 = 2684354560$
u	117	$117 * 27^1 = 3159$	$117 * 32^4 = 122683392$
e	101	$101 * 27^0 = 101$	$101 * 32^3 = 3309568$
r	114		$114 * 32^2 = 116736$
t	116		$116 * 32^1 = 3712$
a	97		$97 * 32^0 = 97$
TOTAL:	**625**	**61580**	**2810468065**

Se puede minimizar el nº de multiplicaciones utilizando la Regla de Horner

$$p(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

puede reescribirse como:

$$p(x) = ((((a_5x + a_4)x + a_3)x + a_2)x + a_1)x + a_0$$

$P*32^5 + u*32^4 + e*32^3 + r*32^2 + t*32^1 + a*32^0$

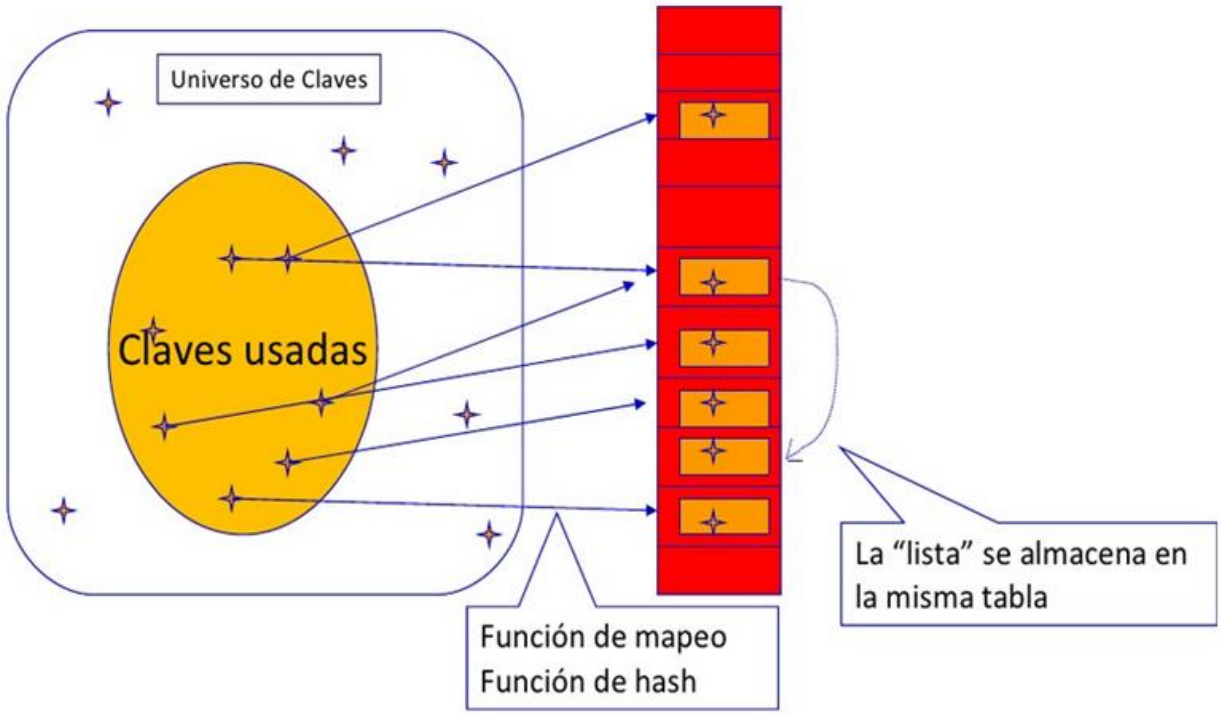
Tablas HASH . Protección Pasiva

Es cuando varios elementos necesitan compartir la misma posición dentro de la tabla

* Varios elementos comparten la misma posición de la tabla hash
  Ej.: cada posición de la tabla es a su vez una lista o un árbol
* Fac**tor de carga** (Load Factor) = n / B, siendo n el número de elementos y B el tamaño de la tabla
  Lo recomendable es que FC <= 1
* Cada posición sólo tiene cabida para un elemento
  Si se detecta una colisión, se buscan posiciones próximas
* Técnicas de búsqueda de posiciones próximas
  - Exploración lineal
  - Exploración cuadrática
  - Dispersión doble

Direccionamiento Abierto o Hash Cerrado

Cuando ocurre una colisión, podemos buscar de manera metódica una nueva posición vacía donde insertar el elemento en lugar de usar la posición generada por la función de hashing. Esta técnica es llamada direccionamiento abierto o hash cerrado.

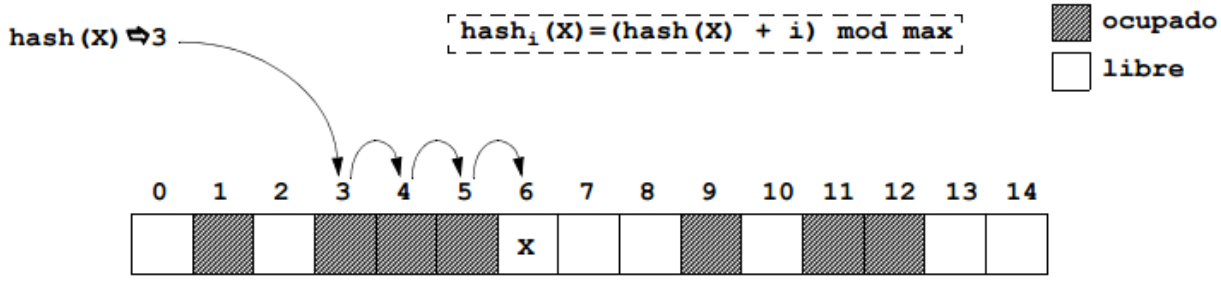


- *Exploración Lineal*

* Se modifica la función f(c) = c % B pasa a ser f(c) = (c + i) % B, siendo i = 0, 1, 2, 3, 4,…

La existencia de agrupamientos provoca problemas

- Cuando se busca un elemento que está en un agrupamiento hay que recorrer todos los elementos del agrupamiento hasta que se encuentra
- Lógicamente, cuando se encuentra una posición vacía no se sigue buscando (podría ser una complejidad O(n)) y se detiene el algoritmo



• Ocupación de la tabla: λ=celdasOcupadas/totalNumCeldas
• Número medio de intentos para encontrar celda libre=1/(1-λ)
- si están ocupadas la mitad de las celdas (λ=0.5) en promedio se necesitan 2 intentos
- Si la ocupación es mayor, el número de intentos crece mucho
• Si la función hash no es homogénea la eficiencia empeora
• Tienden a formarse bloques de celdas ocupadas que degradan la prestaciones

…pero, ¿y si se hubiera borrado algún elemento del agrupamiento?

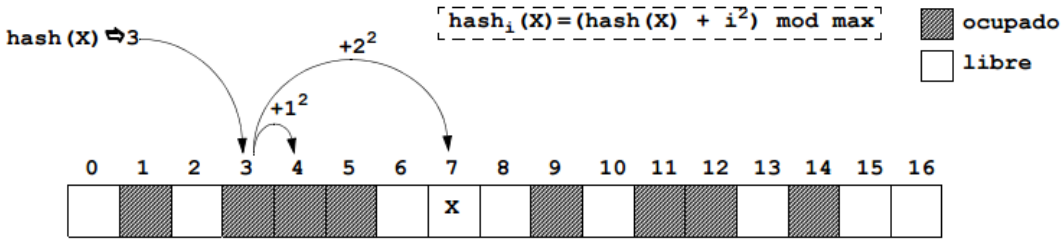
- *Exploración Lineal – Borrado prerezoso*

* No eliminar un elemento hasta que se introduce otro
* Simplemente, se marca
  Para insertar se considera que la posición está libre
  Para buscar se considera que la posición está ocupada

- *Exploración Cuadrática*

Para reducir el número de intentos se necesita un esquema de resolución de conflictos que evite la agrupación primaria. Si la función da como resultado la celda H y está ocupada, se consultan:

$$H + 1^2; H + 2^2; H + 3^2; \ldots ; H + i^2$$



• Evita en gran medida el agrupamiento de celdas ocupadas que ocurría con la exploración lineal
• Es preciso que la ocupación siempre sea inferior a la mitad
• El tamaño de la tabla debe ser un **número primo** para evitar que se formen ciclos en la función que excluyan celdas potencialmente vacías.

Teorema: Dada una tabla de dispersión que utiliza exploración cuadrática, si su tamaño es un número primo siempre podremos insertar un nuevo elemento en la tabla si su ocupación es estrictamente inferior a 0.5. Además, durante el proceso de inserción, no se visita la misma posición más de una vez

- *Dispersión Doble o Doble Hasheo*

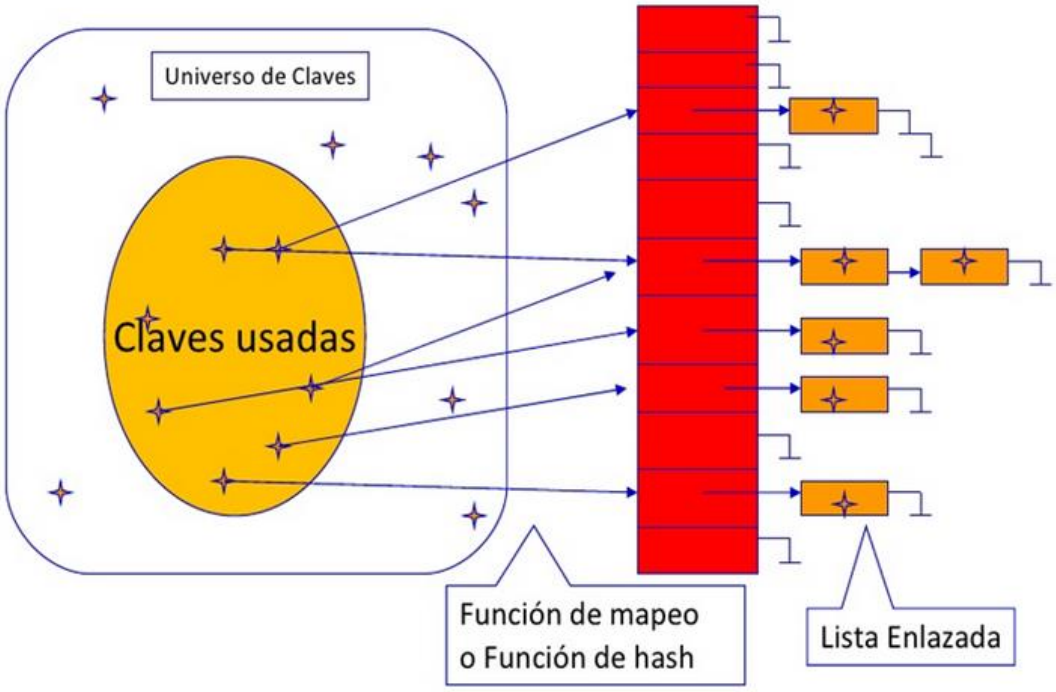
Permite eliminar la agrupación secundaria que se produce en ocasiones con la exploración cuadrática

$$\text{hash}_i(x) = (\text{hash}(x) + i \times \text{hash}_{alt}(x)) \bmod \max$$

$\text{hash}_{alt}(x)$ nunca debe dar cero
- ej.: R-(x mod R), siendo R primo

Tablas Hash – Hash Abierto

Otra técnica que se puede aplicar cuando ocurre una colisión, consiste en crear una lista para cada posición del vector. De esta manera cuando se produce una colisión simplemente se agrega el elemento a la lista. Esta técnica es llamada direccionamiento cerrado o hash abierto.



Resolución por encadenamiento

El peor de los casos para la inserción es O(1). Para la búsqueda, el peor de los casos es proporcional al tamaño de la lista. La eliminación de un elemento x puede ser realizado en O(1) si la lista es doblemente encadenada.

- La inserción se hace como en una pila (al principio) ***con tiempo constante***
- La búsqueda es secuencial .
    - Lo peor que puede pasar es que nuestra función hash mapee concentrados todos los elementos (n) en un solo slot.
    - El caso promedio depende de cuan buena sea nuestra función hash. *(suponiendo que cualquier elemento n tenga la misma probabilidad de ser mapeado a cualquiera de los m slots de T, independientemente de los otros)*..

Para j=0, 1, … , m-1 y sea n_j el largo de la lista T[j], entonces

N= $n_0 + n_1 + n_2 + \ldots + n_j + \ldots + n_{m-1}$

el valor medio es $E[n_j] = \alpha = \frac{n}{m}$

$$\sum_{i=0}^{m-1} p_i = 1 \xrightarrow[\text{Uniform Hashing}]{\text{Simple}} p \underbrace{\sum_{i=0}^{m-1} 1}_{m} = 1 \Rightarrow pm = 1 \Rightarrow p = \frac{1}{m}$$

Teorema 1
*En una tabla hash en la cual las colisiones son resueltas con encadenamiento, una búsqueda **sin éxito** toma un tiempo Θ(1 + α), en promedio, bajo la suposición de hashing uniforme simple.*

Teorema 2
*En una tabla hash en la cual las colisiones son resueltas con encadenamiento, una búsqueda **exitosa** toma un tiempo Θ(1 + α), en promedio, bajo la suposición de hashing uniforme simple.*

• El análisis anterior significa que si el numero de slots, en una tabla hash, es proporcional al numero de elementos en la tabla, se tiene n = O(m) y de esta forma, α = n/m = O(m)/m = O(1).
• Por lo tanto, la búsqueda toma tiempo constante en promedio.
• Como la inserción toma O(1) en el peor de los casos, y el borrado toma O(1) en el peor de los casos cuando las listas están doblemente encadenadas, todas las operaciones pueden tomar tiempo constante en promedio.

Queremos una función que satisfaga "**Simple Uniform Hashing**"
Pero eso no es posible si no conocemos las entradas, ya que puede ser sesgada.

- Método de la división
  h(k)= K mod m $K \in \mathbb{N} + el\ 0$
  Tener una potencia de 2 es deseable porque nos simplifica el calculo pero

  Necesitamos que m sea un numero primo alejado de una potencia de 2
- Método de Multiplicación

  h(k)= ⌊m(KA mod 1)⌋ 0<A<1
  KA mod 1=KA – ⌊KA⌋

  ⌈ ⌉ Trunca al entero superior
  ⌊ ⌋ Trunca al entero inferior

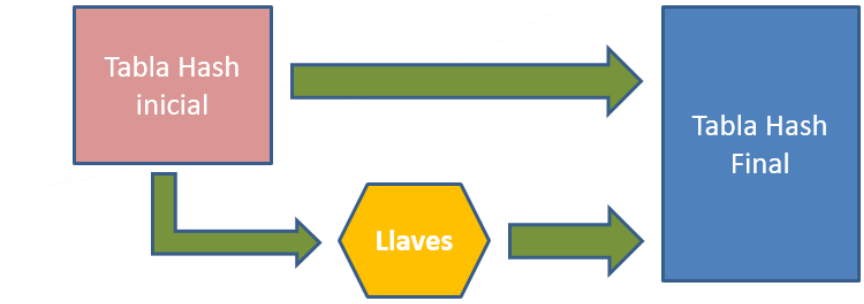
El valor de m no es critico y m es normalmente una potencia de 2

Tablas HASH . Redispersión o Rehash

¿Qué sucede cuando no se puede insertar un elemento debido a que todas las posiciones colisionan?

* Aumentar el tamaño en función del FC
    Tablas hash abiertas à rendimiento decrece si FC > 1
    Tablas hash cerradas à se paraliza si FC > 0,5
* Se busca un nuevo valor B
    El primo inmediatamente superior al tamaño doble del original
    Se recorren los elementos y se añaden a la nueva tabla

Aumentar el tamaño de la tabla y volver a asociar los elementos.



Tablas HASH . Hashing Dinámico

• La función de hash se va modificando de acuerdo al número de elementos y al comportamiento de la tabla.
• Hashing extensible (**Extendible hashing**).
    – Uso de un directorio con M registros.
    – Concatenación de múltiples tablas hash.